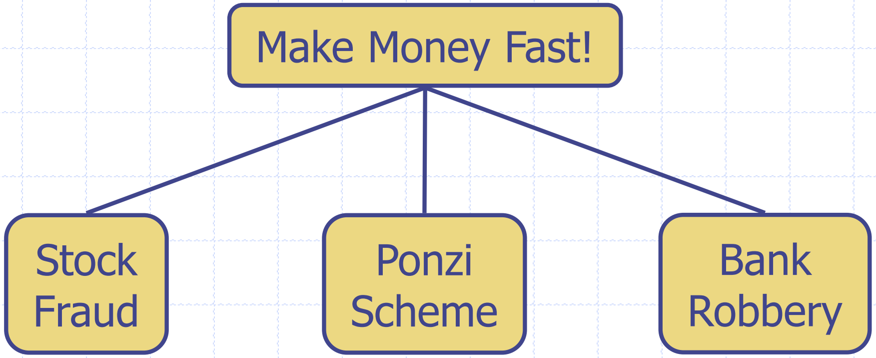
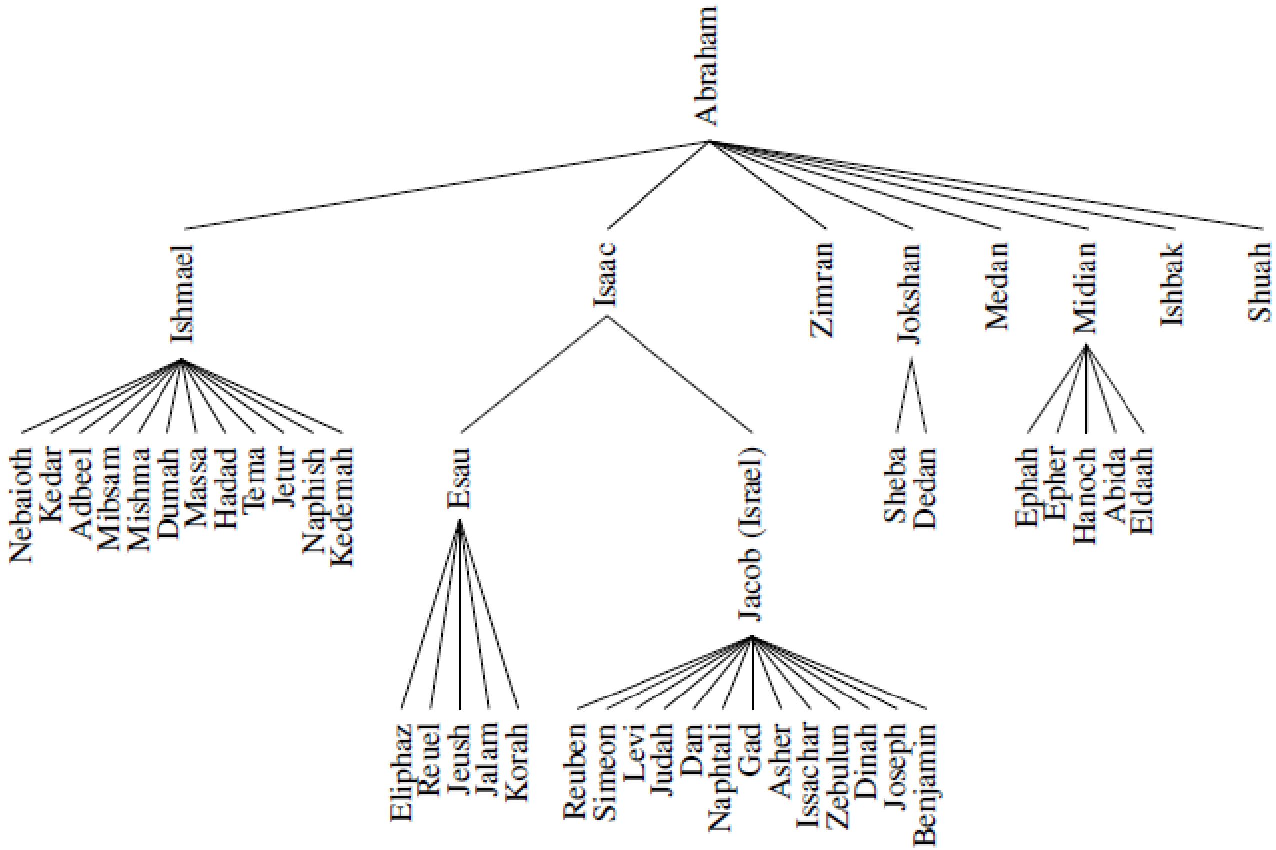
Part 4: Trees

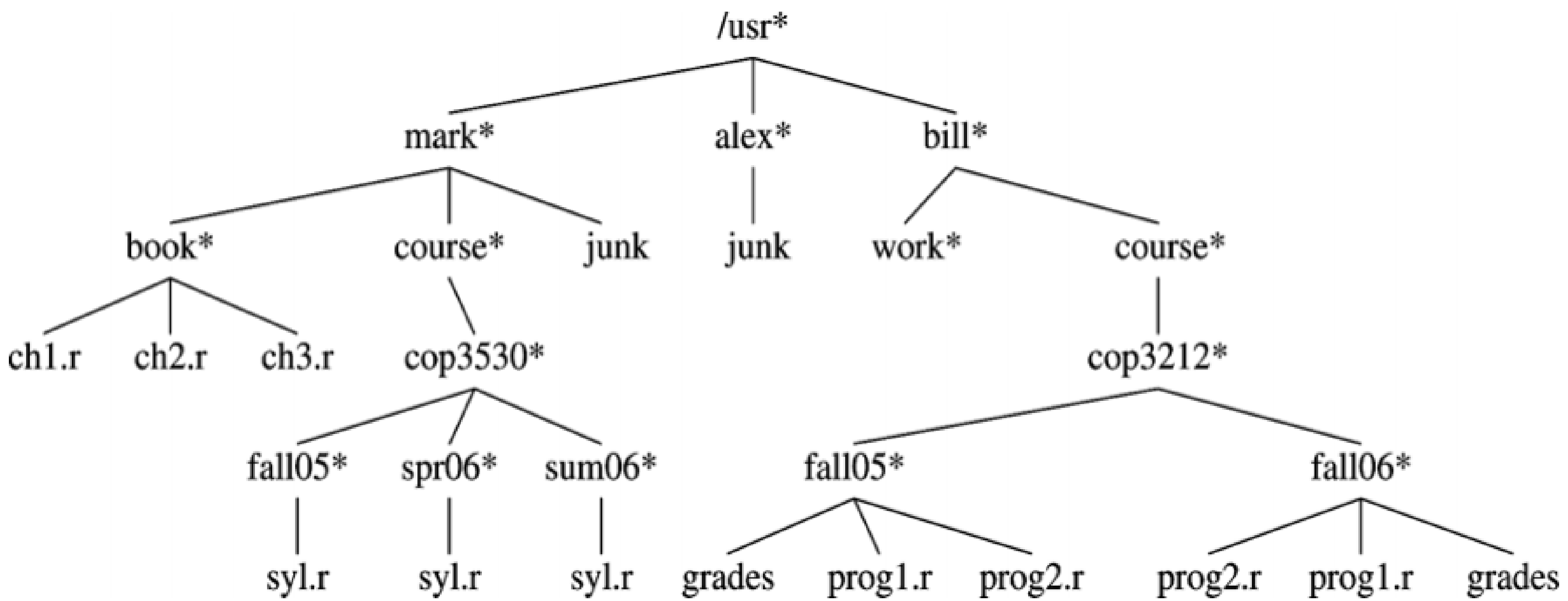
Trees



Example: Family Tree

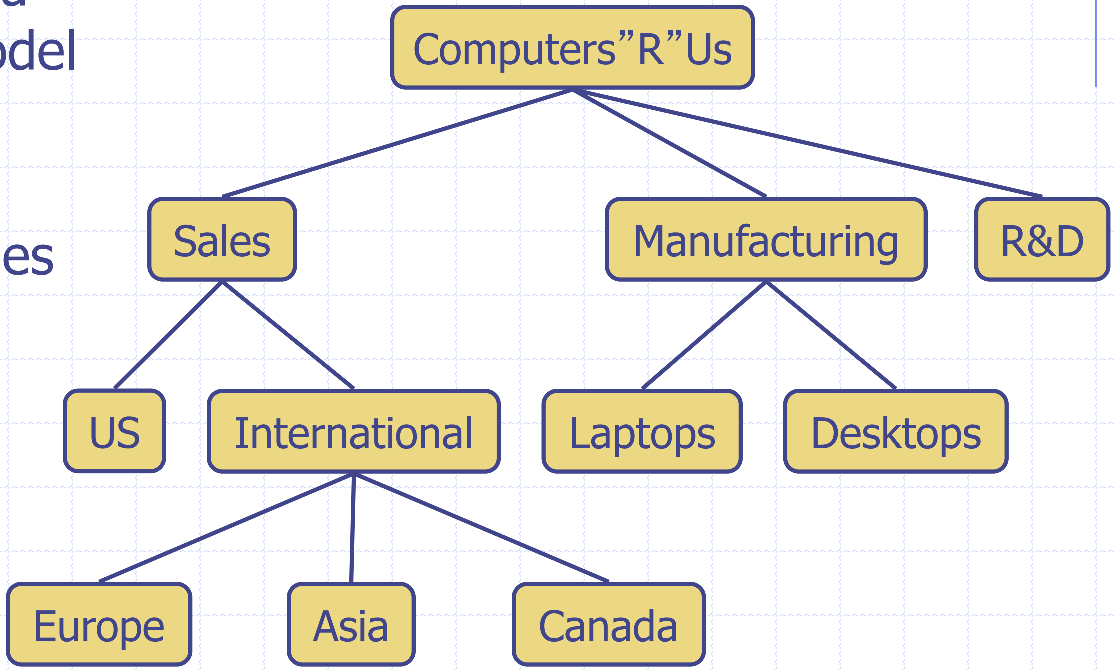


Example: Unix File System

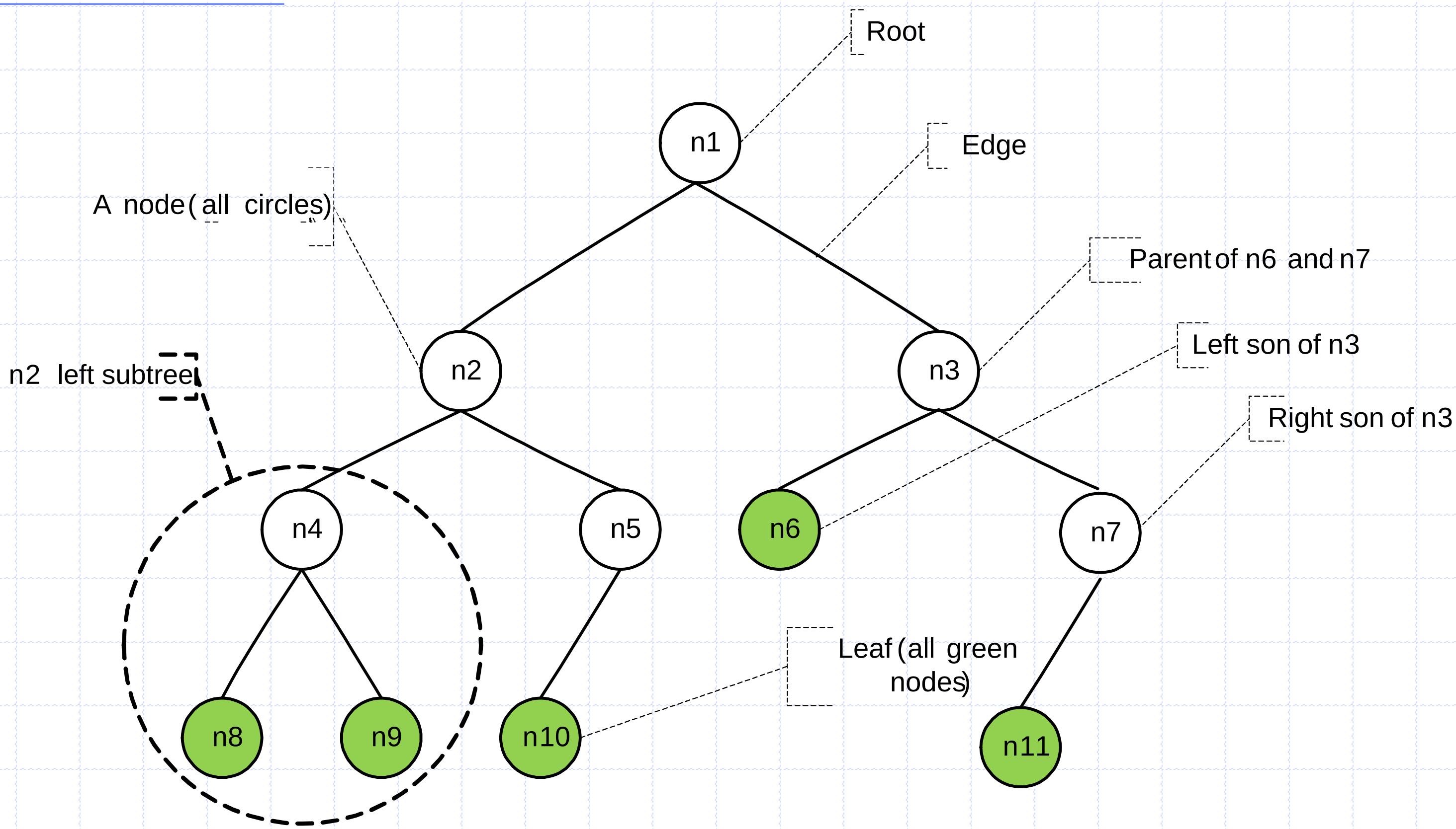


What is a Tree

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments



What is a Tree (Daniel Geva)

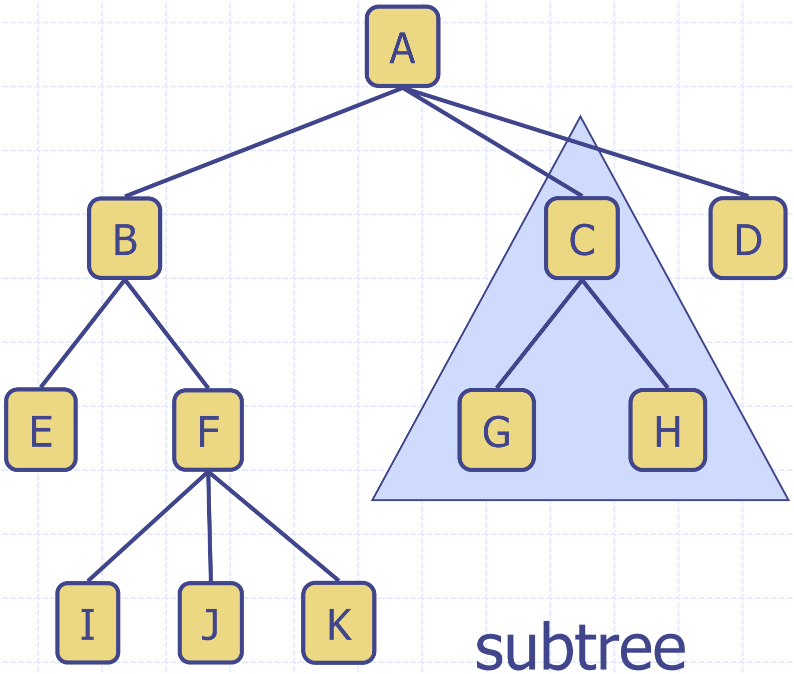


Node height – number of edges on the longest path to a leaf
Tree height – height of the root
Balanced Tree – All non-leaf have two sons

Tree Terminology

- **Root**
  node without parent (A)
- **Internal node**
  node with at least one child (A, B, C, F)
- **Leaf (External node)**
  node without children (E, I, J, K, G, H, D)
- **Ancestors** of a node:
  parent, grandparent, grand-grandparent, etc.
- **Depth of a node:**
  number of ancestors
- **Height of a node:**
  1 + Max height of children
  (leaf height = 0)
- **Height of a tree**
  maximum depth of any node (3)
- **Descendant of a node**
  child, grandchild, grand-grandchild, etc.

- Subtree: tree consisting of a node and its descendants



Tree ADT

- We use positions to abstract nodes, left key is return type:
- Generic methods:
  - Integer len()
  - Boolean is_empty()
  - Iterator positions()
  - Iterator iter()
- Accessor methods:
  - position root()
  - position parent(p)
  - Iterator children(p)
  - Integer num_children(p)

Note: A tree **position** is like a list **index**

- Query methods:
  - Boolean is_leaf(p)
  - Boolean is_root(p)
- Update method:
  - element replace (p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT

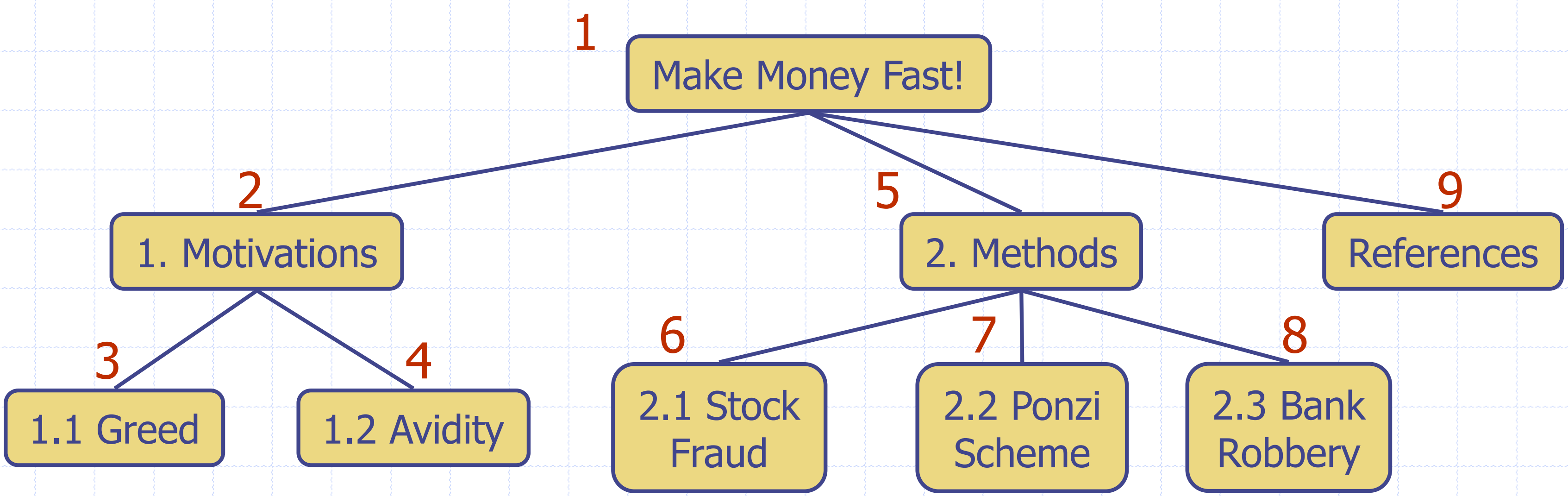
Abstract Tree Class in Python

```
class Tree:
  """Abstract base class representing a tree structure."""

  #------------------------------- nested Position class -------------------------------
  class Position:
    """An abstraction representing the location of a single element."""

    def element(self):
      """Return the element stored at this Position."""
      raise NotImplementedError('must be implemented by subclass')

    def __eq__(self, other):
      """Return True if other Position represents the same location."""
      raise NotImplementedError('must be implemented by subclass')

    def __ne__(self, other):
      """Return True if other does not represent the same location."""
      return not (self == other)            # opposite of __eq__

  # ---------- abstract methods that concrete subclass must support ----------
  def root(self):
    """Return Position representing the tree's root (or None if empty)."""
    raise NotImplementedError('must be implemented by subclass')

  def parent(self, p):
    """Return Position representing p's parent (or None if p is root)."""
    raise NotImplementedError('must be implemented by subclass')

  def num_children(self, p):
    """Return the number of children that Position p has."""
    raise NotImplementedError('must be implemented by subclass')

  def children(self, p):
    """Generate an iteration of Positions representing p's children."""
    raise NotImplementedError('must be implemented by subclass')

  def __len__(self):
    """Return the total number of elements in the tree."""
    raise NotImplementedError('must be implemented by subclass')

  # ---------- concrete methods implemented in this class ----------
  def is_root(self, p):
    """Return True if Position p represents the root of the tree."""
    return self.root( ) == p

  def is_leaf(self, p):
    """Return True if Position p does not have any children."""
    return self.num_children(p) == 0

  def is_empty(self):
    """Return True if the tree is empty."""
    return len(self) == 0
```

Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

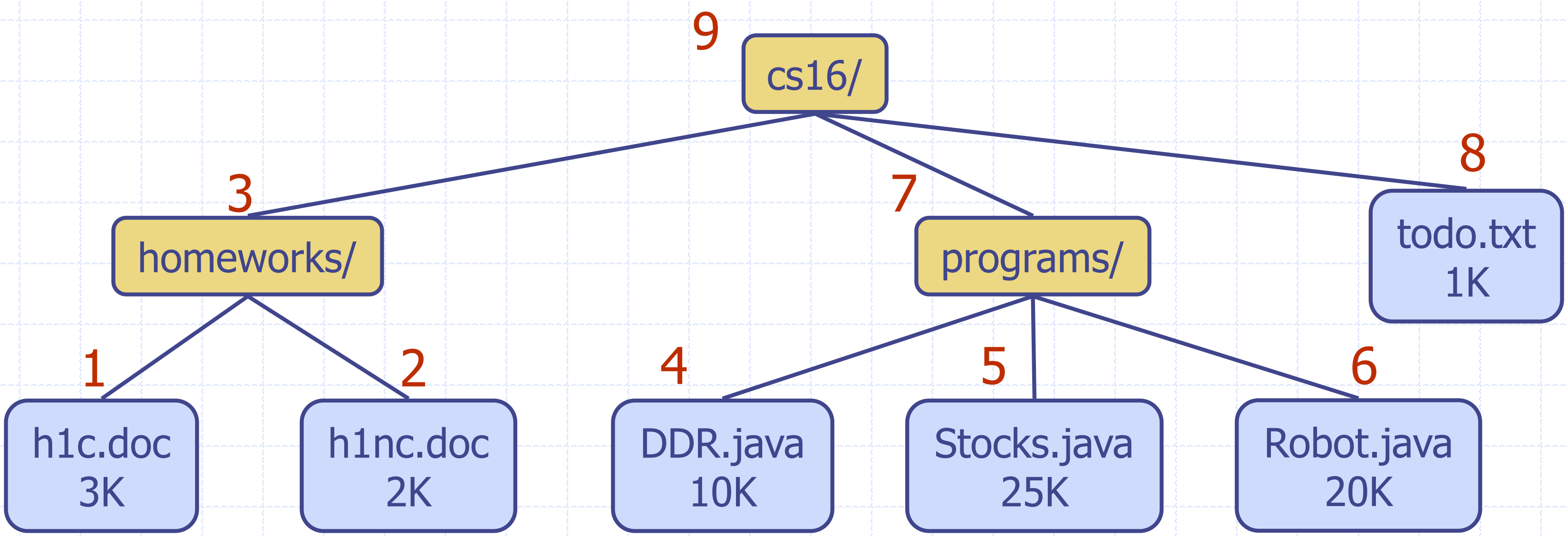
Algorithm *preOrder*(v)
    visit(v)
    for each child w of v
        preOrder (w)



Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

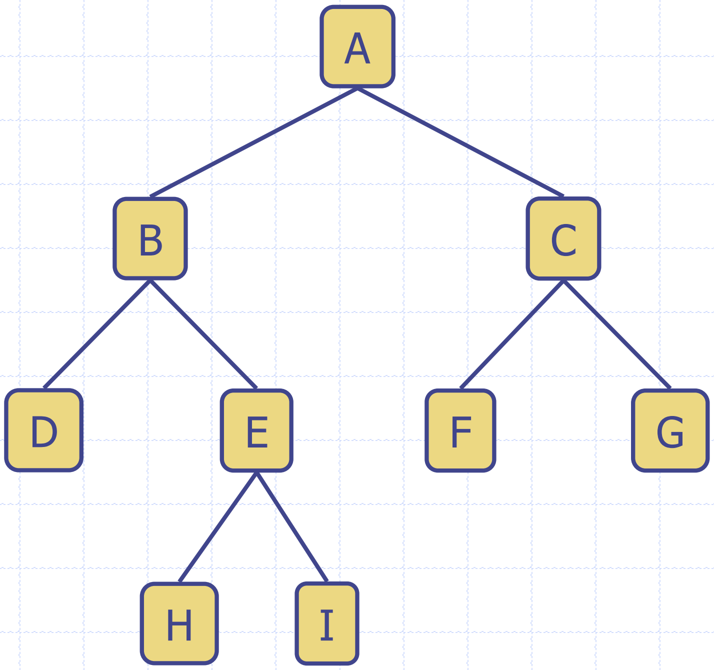
Algorithm *postOrder*(*v*)
    for each child *w* of *v*
        postOrder (*w*)
    visit(*v*)



Binary Trees

- A binary tree is a tree with the following properties:
  - Each internal node has **at most** two children (exactly two for **proper binary trees**)
  - The children of a node are an **ordered pair**
- We call the children of an internal node left child and right child
- Proper Binary Tree: every node is a leaf or must have exactly two children

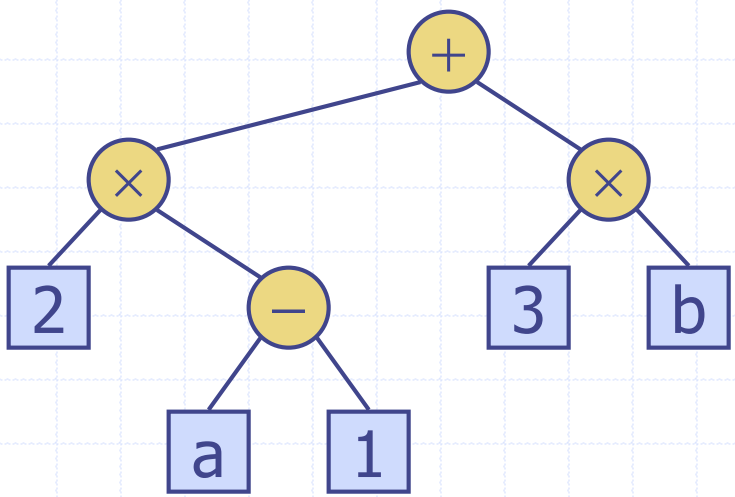
- Applications:
  - arithmetic expressions
  - decision processes
  - searching



LINK TO PYTHON CODE

Arithmetic Expression Tree

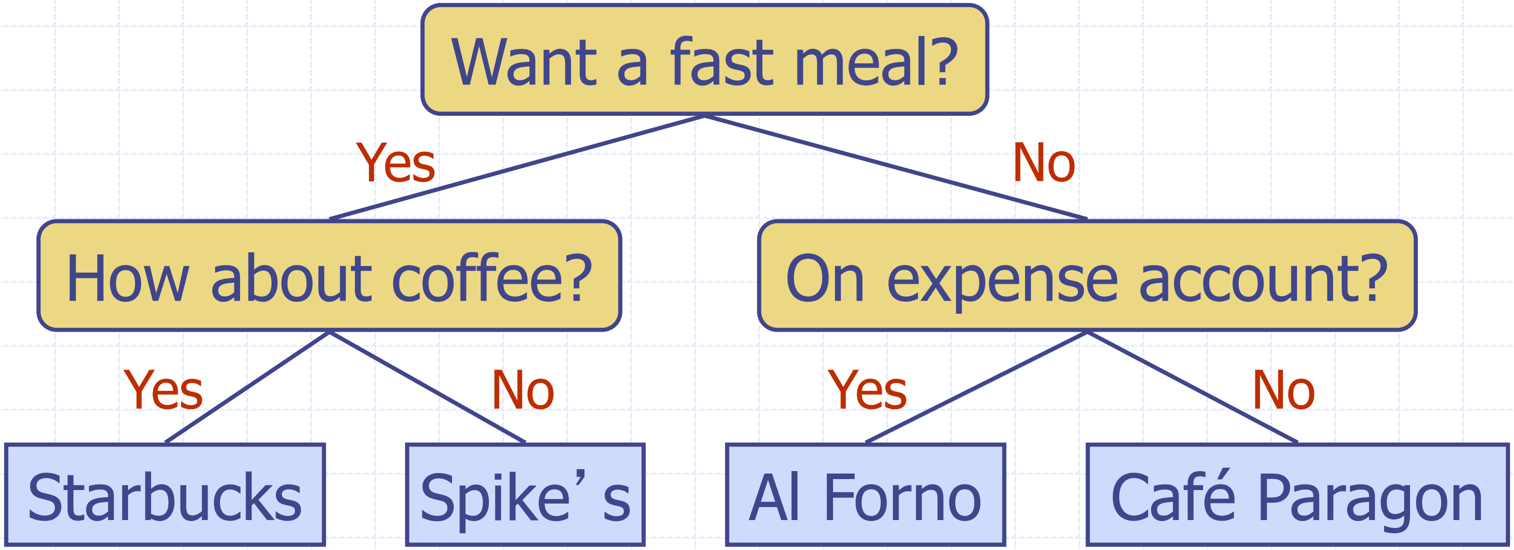
- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$



LINK TO PYTHON CODE

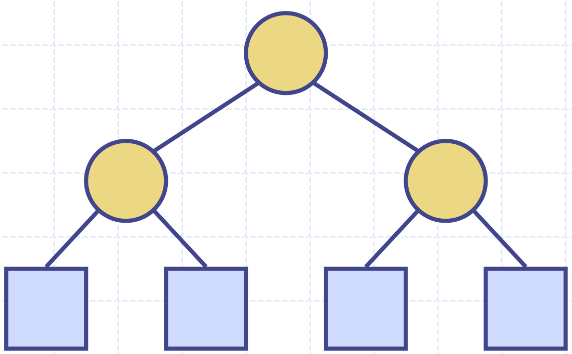
Decision Tree

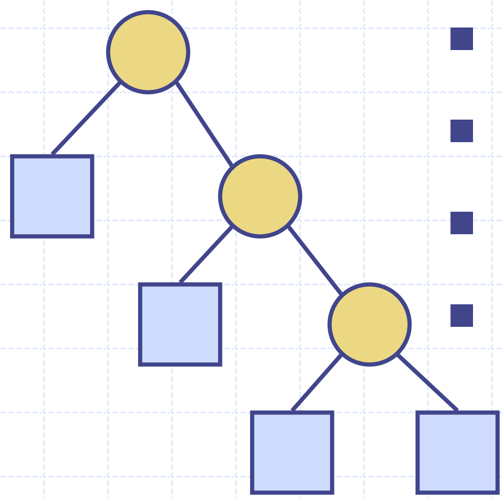
- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision



Properties of Proper Binary Trees

- Notation
  - n number of nodes
  - e number of external nodes
  - i number of internal nodes
  - h height





- Properties:
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2 (n + 1) - 1$

BinaryTree ADT

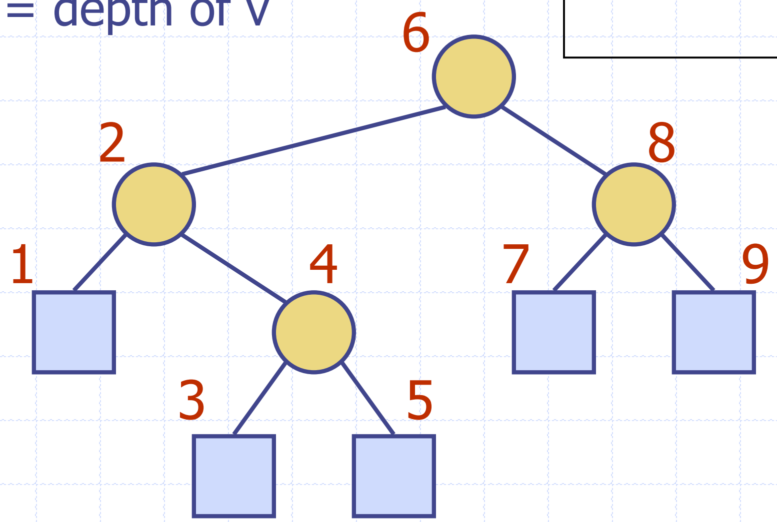
- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
- Additional methods:
  - position left(p)
  - position right(p)
  - position sibling(p)
- Update methods may be defined by data structures implementing the BinaryTree ADT

LINK TO PYTHON CODE

Inorder Traversal

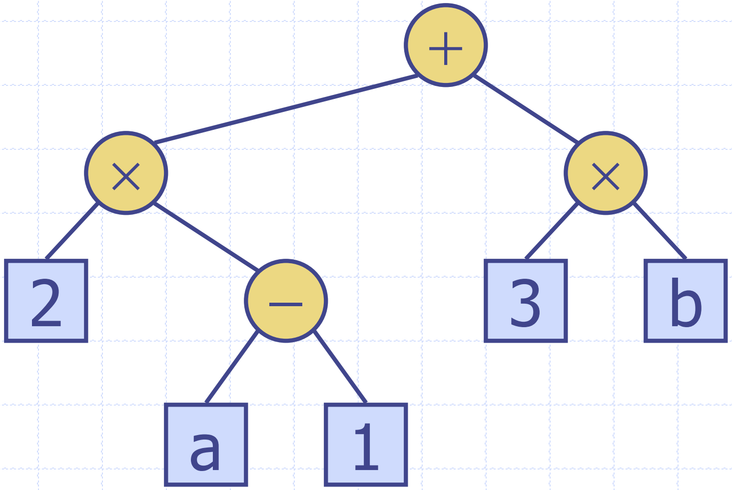
- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
  - x(v) = inorder rank of v
  - y(v) = depth of v

Algorithm *inOrder*(*v*)
    if *v* **has a left child**
        inOrder (*left* (*v*))
    visit(*v*)
    if *v* **has a right child**
        inOrder (*right* (*v*))



Print Arithmetic Expressions

- Specialization of an inorder traversal
  - print operand or operator when visiting node
  - print "(" before traversing left subtree
  - print ")" after traversing right subtree



Algorithm *printExpression*(*v*)
  if *v* **has a left child**
    print("(")
    inOrder (*left*(*v*))
  print(*v.element* ())
  if *v* **has a right child**
    inOrder (*right*(*v*))
    print (")")

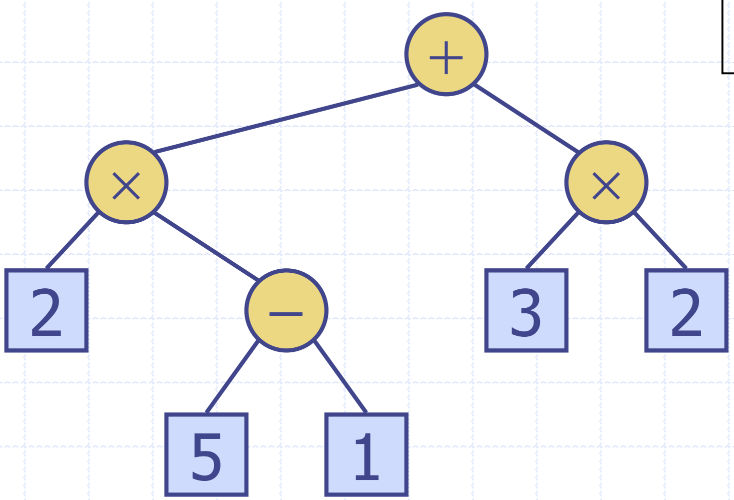
$((2 \times (a - 1)) + (3 \times b))$

LINK TO PYTHON CODE

Evaluate Arithmetic Expressions

- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

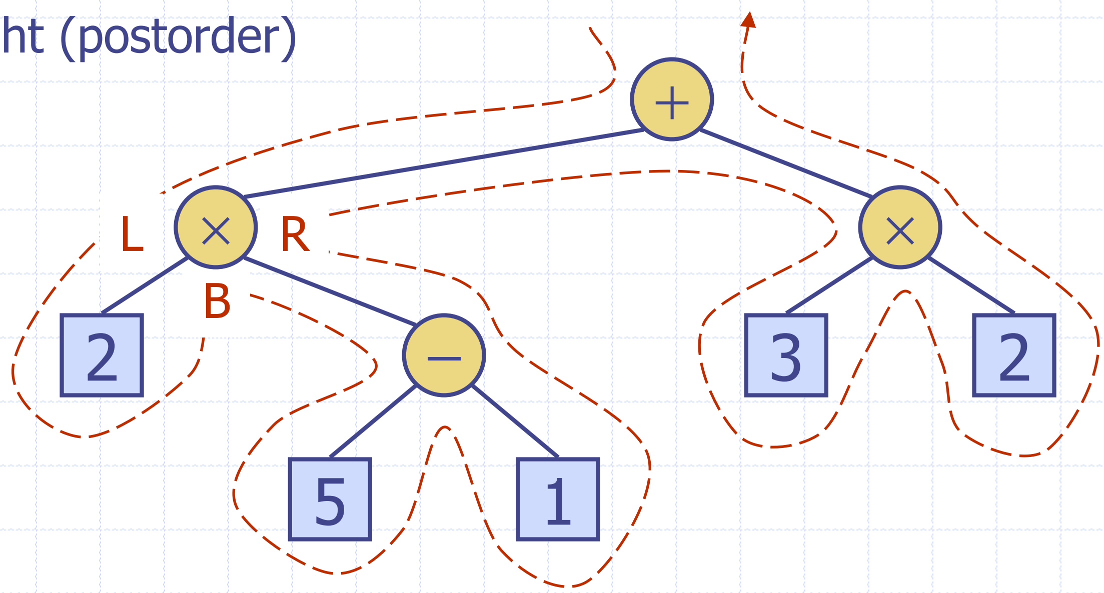
Algorithm *evalExpr*(v)
    if *is_leaf* (v)
        return *v.element* ()
    else
        $x \leftarrow$ *evalExpr*(*left* (v))
        $y \leftarrow$ *evalExpr*(*right* (v))
        $\Diamond \leftarrow$ operator stored at v
        return $x \Diamond y$



LINK TO PYTHON CODE

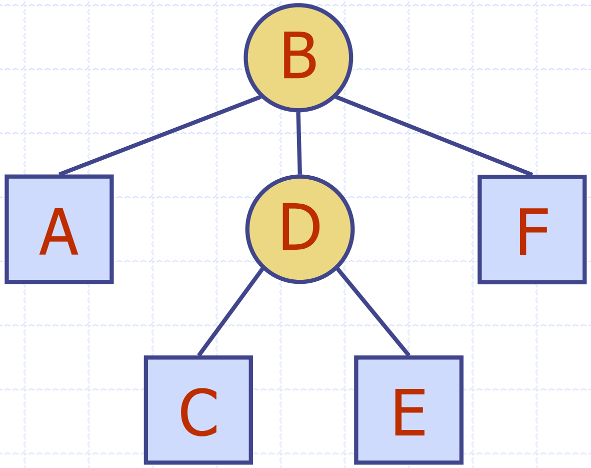
Euler Tour Traversal

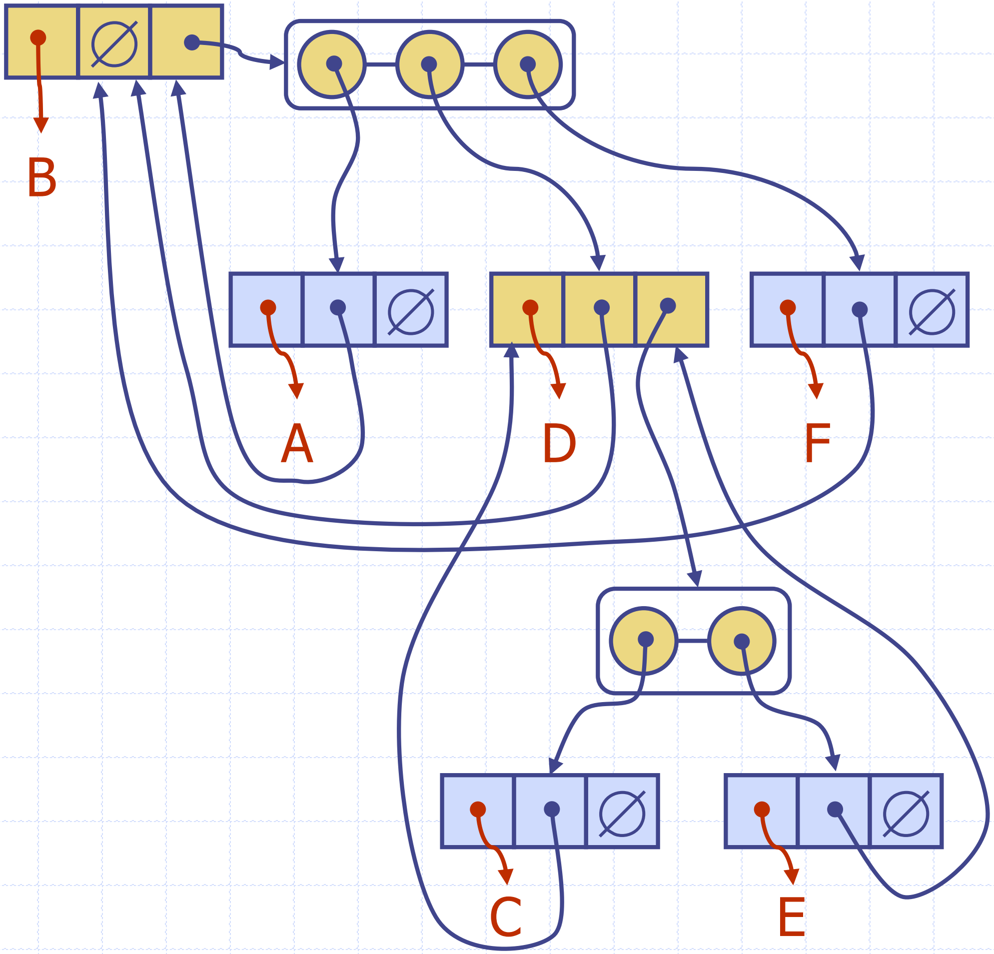
- Generic traversal of a binary tree
- Includes a special cases the preorder, postorder and inorder traversals
- Walk around the tree and visit each node three times:
  - on the left (preorder)
  - from below (inorder)
  - on the right (postorder)



Linked Structure for Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes
- Node objects implement the Position ADT



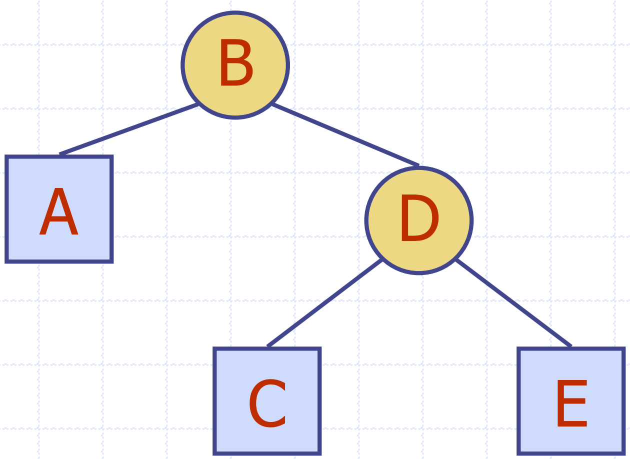


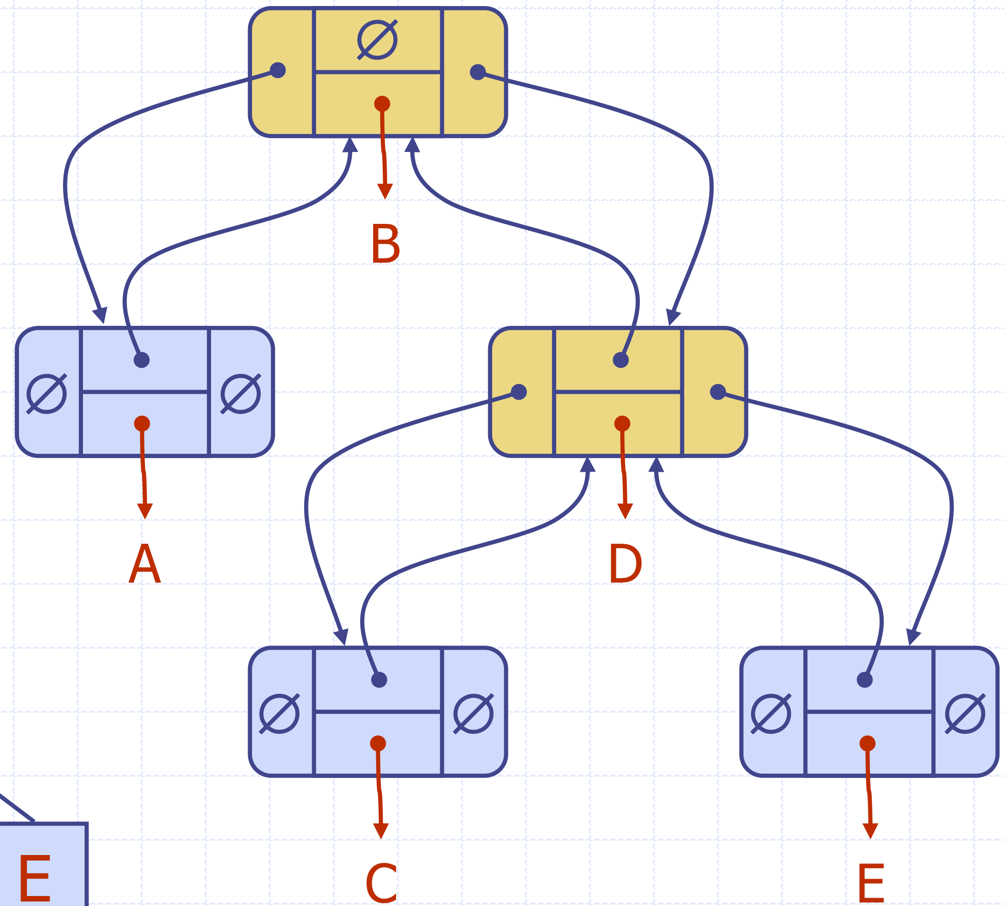
The Node Class

```
class Node:
    "Class for storing a binary tree node"

    def __init__(self, element, parent=None, left=None, right=None):
        self.element = element
        self.parent = parent
        self.left = left
        self.right = right
```

Linked Structure for Binary Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node
- Node objects implement the Position ADT

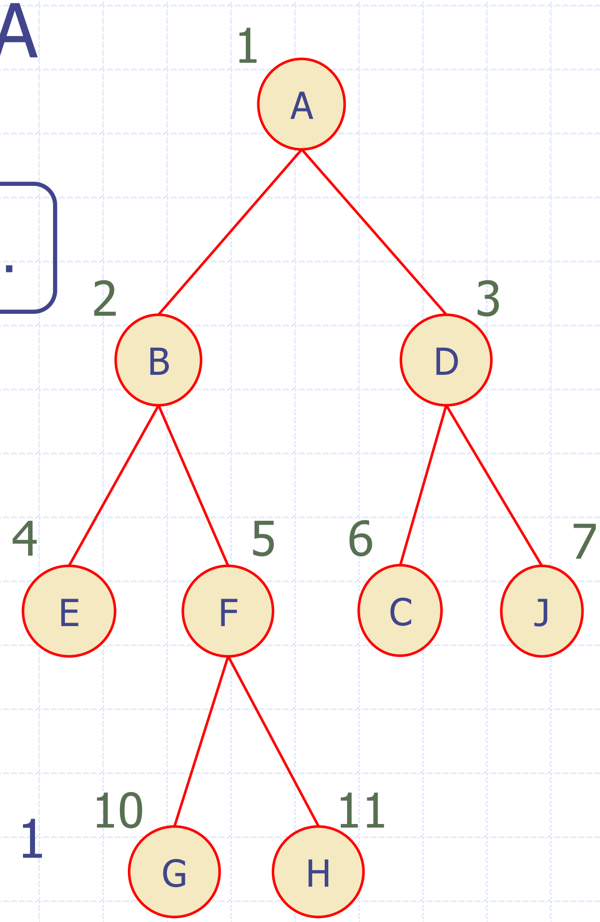




Array-Based Representation of Binary Trees

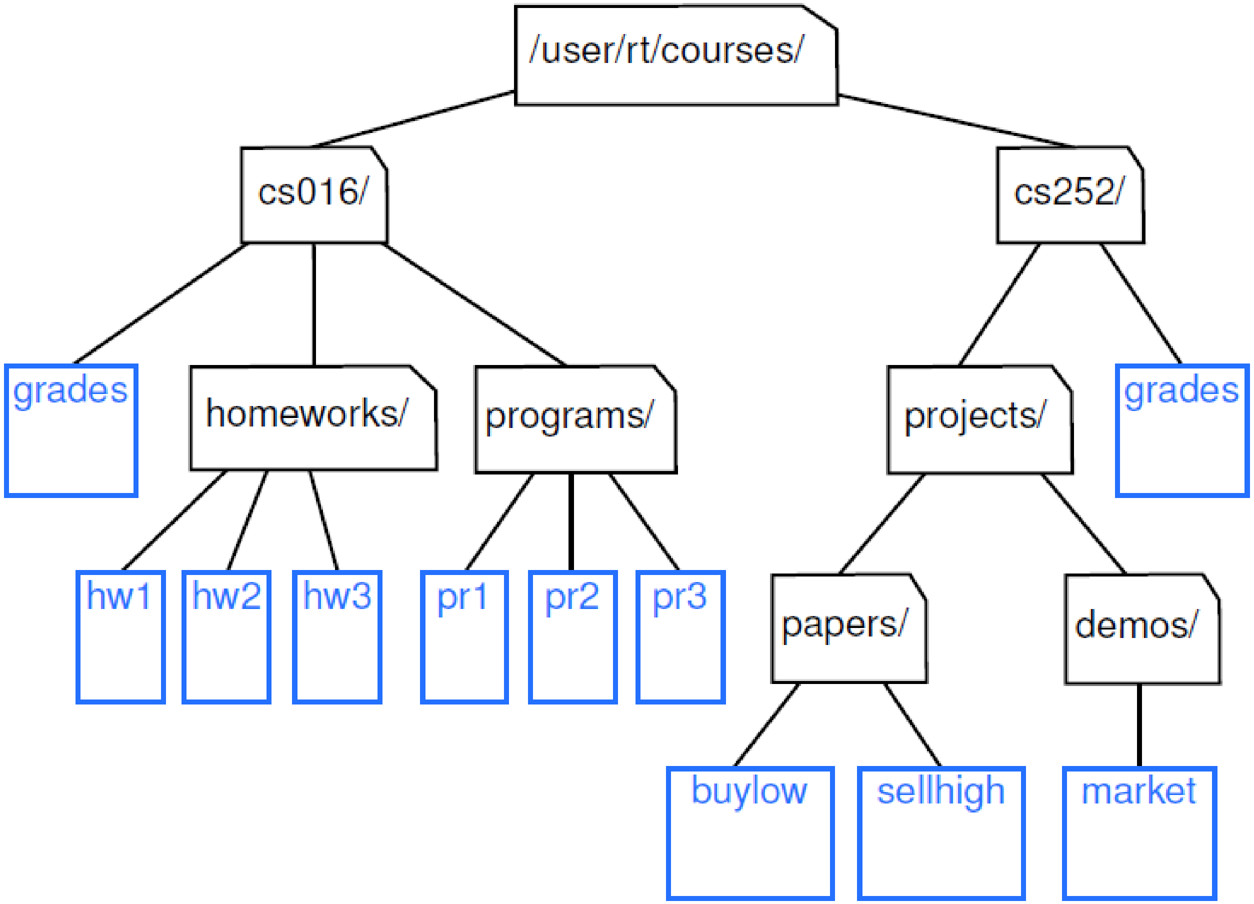
- Nodes are stored in an array A

	A	B	D	…	G	H	…
0	1	2	3		10	11	



- Node v is stored at A[rank(v)]
  - rank(root) = 1
  - if node is the left child of parent(node), rank(node) = $2 \cdot$ rank(parent(node))
  - if node is the right child of parent(node), rank(node) = $2 \cdot$ rank(parent(node)) + 1

Example: Directory Disk Space



Example: Directory Disk Space

```
import os

def disk_space(dir):
    size = 0
    for file in os.listdir(dir) :
        path = dir + "/" + file
        if os.path.isfile(path):
            size += os.path.getsize(path)
        else:
            size += disk_space(path)
    return size
```